Leaving Stragglers at the Window: Low-Latency Stream Sampling with Accuracy Guarantees

Omar Farhat, Harsh Bindra, Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
{obfarhat,hbindra,kdaudjee}@uwaterloo.ca

ABSTRACT

Stream Processing Engines (SPEs) are used to process large volumes of application data to emit high velocity output. Under high load, SPEs aim to minimize output latency by leveraging sample processing for many applications that can tolerate approximate results. Sample processing limits input to only a subset of events such that the sample is statistically representative of the input while ensuring output accuracy guarantees. For queries containing window operators, sample processing continuously samples events until all events relevant to the window operator have been ingested. However, events can suffer from large ingestion delays due to long or bursty network latencies. This leads to stragglers that are events generated within the window's timeline but are delayed beyond the window's deadline. Window computations that account for stragglers can add significant latency while providing inconsequential accuracy improvement. We propose Aion, an algorithm that utilizes sampling to provide approximate answers with low latency by minimizing the effect of stragglers. Aion quickly processes the window to minimize output latency while still achieving high accuracy guarantees. We implement Aion in Apache Flink and show using benchmark workloads that Aion reduces stream output latency by up to 85% while providing 95% accuracy guarantees.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

Stream processing, Sampling, Windows, Watermark

ACM Reference Format:
Omar Farhat, Harsh Bindra, Khuzaima Daudjee. 2020. Leaving Stragglers at the Window: Low-Latency Stream Sampling with Accuracy Guarantees. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3401025.3401732

1 INTRODUCTION

Streaming systems are the primary solution for applications characterized by the need to process large volumes of data in high-velocity. Examples of such modern applications include high-frequency trading [15, 39], network traffic [43], environment monitoring [7, 32], and others [1]. Stream processing engines (SPEs) typically process the deployed *streaming queries* by leveraging parallelization of *operator* instances across the available computational resources. Existing SPEs such as Apache's Flink [8], Spark [45], and Storm [42] fulfill real-time requirements [39] by striving to deliver rapid responses. However, as the load of *events* increases beyond the capacity of the resources, SPEs struggle to maintain the desired performance goals.

In SPEs, the problem of query processing is of significant importance as the substantial cost of processing high volume of events violates the real-time requirement [39]. *Sample processing* is a computing paradigm proposed to enforce this requirement by efficiently processing queries via limiting the input size to a subset of events [27, 34]. Fundamentally, it achieves efficiency by trading-off output accuracy for lower latency. This trade-off is viable for many streaming applications as timely generated output with accuracy guarantees is often much more useful than latent or delayed output with exact accuracy [4, 7, 19, 24, 25].

Streaming queries popularly utilize sample processing to reduce the processing cost of events. For queries exhibiting window operators, sample processing initially selects a subset of incoming events such that processing them would satisfy the output accuracy requirements. At the time of window completion, that is, after all the events relevant to the window operator have been observed by the SPE, the sample is then processed downstream to the output operator. In SPEs, input completion is signaled by *watermarks* [28, 38], which are widely used marker events that signal to window operators that they can process the input. Watermarks are injected into the stream to signify that no further events are expected beyond a designated timestamp.

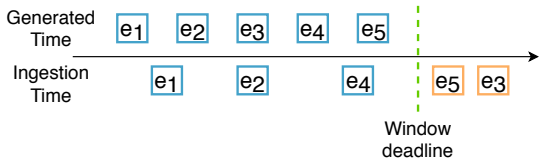


Figure 1: Example illustrating the difference between generation and ingestion time for each event. On-time events are highlighted in blue, stragglers are highlighted in orange.

In sample processing, the output is typically propagated downstream only *after* input completion. SPEs can experience long wait times for input completion for a window due to network delays

DEBS '20, July 13–17, 2020, Virtual Event, QC, Canada

ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401732

[33, 36]. For instance, in the example illustrated in Fig.1, the SPE waits for the arrival of events e_3 and e_5 even after the window's deadline. Consequently, *stragglers* delay input completion thereby increasing output latency. Consider Fig. 2 that illustrates the impact of stragglers on output latency for a time-window of size 1.5s where the network delay of events varies based on distributions that are modeled from real-world traces [36]. For a network delay that is Gamma distributed (light-tail), stragglers impose a significant 25% additional wait-time to guarantee input completion. As for Exponentially distributed delays (heavy tail), the imposed latency is exacerbated by more than 99%, effectively delaying the input completion way beyond the window's deadline. The impact of stragglers on output latency is significantly high so as to overshadow the acquired benefits from sample processing. To the best of our knowledge, none of the existing sample processing techniques mitigate the impact of stragglers on the output latency [14, 17, 25].

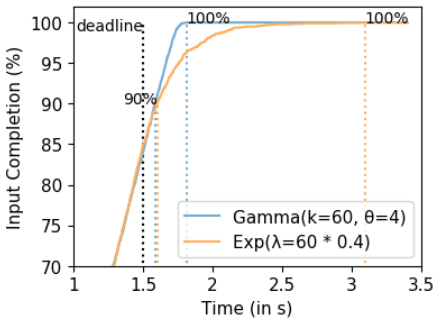


Figure 2: Input completion rate for window operators with network delay for events modeled by empirically verified distributions.

Mitigating the problem of stragglers on sample processing while striking a balance between accuracy and latency is a challenging problem to solve. This is because:

- **(i)** Straggler count and delay patterns vary based on an application's environment. To choose a sample that satisfies the specified accuracy guarantees requires constructing reliable estimations of the straggler events. However, network delays, and in particular the case of exponential delay distributions, adds high variability to any estimation technique.
- **(ii)** Choosing a sample that satisfies the accuracy guarantees is not a trivial task. The sampled events need to be statistically representative of the original input that includes the stragglers. Furthermore, the size of the sample needs to be intelligently chosen based on the functionality of the window operator.
- **(iii)** Determining the minimum number of stragglers to include in the sample can have a large impact on the output latency. As illustrated in Fig. 2, stragglers impose a significant delay penalty on the output latency. Hence, the number of included stragglers needs to be minimized.

In this paper, we demonstrate that sample processing does not need to wait for input completion to process its sampled input. More specifically, existing sample processing approaches do not need to add a *slack* delay to account for stragglers [24]. Instead, windows can propagate their output downstream as soon as output accuracy requirements are satisfied thereby circumventing the costly slack delay, and, consequently, reducing the output latency significantly.

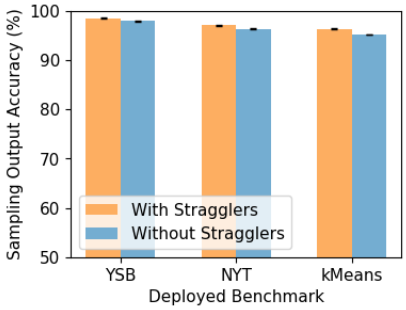


Figure 3: Relative accuracy of sample processing window operators with and without stragglers.

To illustrate the impact of stragglers on output results, consider Fig. 3 that shows output accuracy results obtained running a sample processing algorithm in two settings: with and without stragglers. The first setting takes stragglers into account thereby waiting for input completion while the second circumvents stragglers by processing the window's input as soon the deadline is due and the accuracy requirements are satisfied. In both settings, the algorithm ran with a target output accuracy of 95% over two popular streaming benchmarks that include windowed operators of different functionalities. The first popular benchmark is the Yahoo! Streaming Benchmark (YSB) [20] and the second is the New York Taxi (NYT) benchmark [32]. The figure shows that sampling with stragglers provides an insignificant improvement of *less than* 1% compared to sampling without stragglers for both YSB and NYT benchmarks. We also ran the kMeans benchmark as an example of a query with a windowed operator of higher complexity. The accuracy difference between the two samples was only about 1% indicating that the two samples shared identical statistical significance. These results demonstrate that not only do stragglers impose *large* output latency on input completion, they contribute *insignificantly* towards achieving higher output accuracy.

This observation motivates the work in this paper in which we present the design and implementation of our sample processing algorithm called Aion. Aion continuously monitors and samples important patterns in the workload such as network and inter-event generation delays to estimate the pattern of events and stragglers. Aion then utilizes these estimations, in addition to the type of the window operator, to compute the minimum sample size that achieves the accuracy guarantees. Aion also exploits straggler patterns by *intelligently* processing the sample before input completion such that the impact of stragglers is mitigated. As we demonstrate in our experiments, Aion delivers significant performance gains to reduce latency by as much as 80%.

This paper is organized as follows: We describe background material in Sec. 2 and Aion's design and its algorithmic details in

Sec. 3. We present our experiments in Sec. 4, discuss relevant work in Sec. 5, and conclude in Sec. 6.

2 BACKGROUND

This section discusses windows, watermarks, and sample processing to provide relevant background on the problem, after which we present our design of Aion.

2.1 Windows and Watermarks

Windows are a construct used for grouping events together which exhibit similar properties. SPEs typically use windows to group events on which to execute join, aggregation and selections. Windows enable flexibility and allow complex grouping selections [21]. Common constructs are time based and sequence based windows. For example, a time-based window can group all events generated in the last ten seconds while a sequence based window will group the next fifty events received.

A component of a window is its *deadline*. An event is associated with two timestamps; the timestamp at which it is generated at the source and the timestamp at which it is received by the SPE. The deadline represents the cut-off time for an event to be a member of a window according to its generated time. For example, the deadline of a five-second tumbling time-window starting from timestamp zero has a deadline at five seconds, at ten seconds and at subsequent multiples of the five second deadline. As for a sliding five-second time window with a slide of one, the sequence of deadlines starting from the first deadline is (at) five seconds, six seconds, seven seconds, and at subsequent increments of one second.

Stragglers, illustrated in orange in Fig. 1, are of primary interest in this paper. A straggler is an event which belongs in a window operator's computation according to its generated time but arrives *after* a window's deadline. In theory, a straggler could be delayed for an unbounded amount of time and so a decision must be made of when to stop waiting for events and process the window; this decision is triggered by an event called a watermark. Deciding on when to emit a watermark can be a non-trivial problem since cutting the window too short can induce the window operator to process an insufficient number of events and lead to incorrect output.

Watermarks serve as a contract between the user and the SPE that strives for input completeness as well as output correctness. Watermarks can be injected into the stream by the source or an operator which periodically emits watermarks [8]. Methods of generating watermarks can be application specific but typically watermarks are propagated periodically. The value of the watermark, say t, informs the application that it has seen all events with timestamp at most t. For example, a watermark with a period of three seconds can be generated every three seconds, each one holding an increasing value of $t, t+3, t+6, \ldots$. An example is provided in Fig. 4 where upon receiving a watermark with value 3, the events in green (with $ts \leq 3$) are processed. The green timestamp 2 is dropped since it arrives after its corresponding (green) watermark. Upon receiving the blue watermark with value 6, blue events (with $ts \leq 6$) are processed. In this paper, we design an algorithm that automates the generation of watermarks based on the workload properties. We discuss this further in Sec. 3.

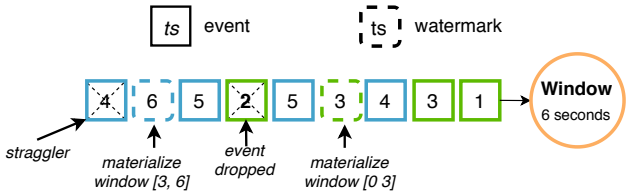


Figure 4: Example illustrating the concept of watermarks in SPEs. Events are consumed in order starting from right to left and each event holds its generation timestamp at the source. Events that are of the same colour as a watermark are processed with the ingestion of that watermark.

Importantly, the watermark's spawning frequency can be used to determine *progress of the stream*. By continuously receiving watermarks, window operators can estimate their input completion percentage. In the example of Fig. 4, consider a window of size 6 seconds. A watermark of value 3 indicates to the window operator that it has seen half of its expected input.

2.2 Approximate Query Processing

Approximate Query Processing (AQP) techniques generally applied on the windowed operators in streaming queries offer a trade-off on accuracy to optimize for specific performance goals. There exist multiple ways in which AQP achieves this balance.

Sketches [9, 13, 37] aim to minimize the memory footprint of stateful operators like windows. Sketches utilize complex data structures that maintain statistically representative information of the input. The stored information is then leveraged to approximate the window's output. However, sketches are not designed to reduce output latency as the entailed complex processing can significantly add to the processing cost. Since we are interested in reducing the output latency, the use of sketches is out of scope for this work.

Sample Processing [24, 31, 35, 40, 41] is an AQP technique that aims to minimize the output latency by rapidly producing an output with accuracy guarantees. Sample processing achieves its goal by limiting its input only to subset of events such that the sample is statistically representative of the input to ensure output accuracy guarantees. In doing so, existing sample processing techniques experience high output latency delay to account for stragglers before emitting an output.

Aion is designed as a sample processing technique since we are interested in the problem of reducing output latency through mitigating the impact of stragglers.

2.3 Sample Processing

Sample processing techniques select a sample that is a subset of events such that the sample is small enough in size to reduce the processing cost while being sufficiently representative of the original population to achieve the specified accuracy guarantees.

We leverage *Bernoulli sampling* in the design of Aion. Bernoulli sampling determines whether an event becomes part of the sample with probability $\theta \in (0, 1)$. This sampling guarantees that (i) all events have an equal probability, θ, of being included in the sample, and (ii) the sample size is a fraction, θ, of the original input size.

Bernoulli sampling can be employed in applications where the input size is unknown as the technique always produces a sample of size proportional to the original input. Due to these properties, Bernoulli sampling integrates well into SPEs where the stream size and input arrival rate are unknown. Aion leverages Bernoulli sampling to build a sample that is statistically representative of the original input while being sufficiently small to reduce the processing cost.

3 AION: STRAGGLER-FREE SAMPLING

We now present the design of Aion including its algorithmic details. Fundamentally, in the context of windowed streams, Aion's main objective is to collect a sample of minimal size such that processing this sample produces an output that is within a specified error threshold r_{thr} of the exact output. We formally express the error by $P(r \leq r_{thr}) \geq 1-\delta$, where r refers to the relative error discrepancy obtained by processing the original and the sampled inputs, and 1 - δ represents the probability of obtaining an error less than r_{thr}. The sample needs to be carefully chosen such that its size is minimal so as to reduce its processing cost. Importantly, the sample size and distribution of its values need to be sufficiently representative of the original input to satisfy the accuracy requirements.

Traditional sample processing techniques complete sampling their input only after the stream consumes a watermark. However, since watermarks signal input completion thereby accounting for stragglers, a significant output latency can be imposed. For instance, [33] presented an algorithm that mandates all events to *slack* for k-seconds before being processed, where k is continuously adjusted to the maximum observed network delay value. Stragglers, however, contribute minimally in improving the accuracy of the sample (Fig. 3).

To circumvent the effects of stragglers on output latency, Aion leverages control over stream progress by automating the generation of watermarks. Watermarks divide a stream into *sub-streams*, each of which is defined over a periodic time-range. After watermark ingestion by the window operator, all events of prior timestamps consumed as part of sub-streams can then be safely processed. Therefore, Aion generates watermarks frequently to ensure incremental processing by window operators [5, 28]. Aion minimizes the impact of stragglers on output latency by generating a watermark as soon as the sampling requirements over each sub-stream are satisfied, even if all stragglers have not yet been ingested.

Aion's design inherently supports incremental processing by sampling from each sub-stream at a customized rate. This strategy ensures more accurate estimations (for network and inter-event generation delays) since the workload data distribution has a lower likelihood of changing within each sub-stream. As soon as the accuracy requirements are achieved, potentially before input completion, Aion generates a watermark to process the sub-stream at the window operator, effectively circumventing the effects of stragglers. The length of each sub-stream f (in milliseconds), also defined as the periodicity of watermarks, is essential to the algorithm's performance. A smaller value of f ensures higher uniformity for the input rate of each sub-stream at the expense of higher algorithm overhead. On the other hand, a larger value of f benefits from a lower overhead but imposes higher likelihood of input rate fluctuation. Aion is designed to leverage the granularity of f to proactively fine tune its input rate anticipation over the upcoming sub-streams. In our experimentation section, we choose values of f that empirically struck the best balance.

Symbols	Definitions
r_{thr}	User defined error margin
r	True error margin
δ	Probability of obtaining r below r_{thr}
f	Defined length for every sub-stream
n^w	Target Sample size over the windowed stream w
n_i^w	Number of events sampled over sub-stream i in stream w
d_i^w	Network delays over the i^{th} sub-stream in the windowed stream w
g_i^w	Inter-event generation delays over the i^{th} sub-stream in stream w
v_i^w	Event values over the i^{th} sub-stream in stream w
D_i^w	Random variable defined over the distribution of d_i^w
G_i^w	Random variable defined over the distribution of g_i^w
V_i^w	Random variable defined over the distribution of v_i^w
m	History size considered by the random distributions D_i^w, G_i^w, and V_i^w
N^w, N_i^w	Number of events received in stream w, and sub-stream i in stream w, respectively
n_i^w	Total number of events sampled over sub-stream i in stream w
s_i^w	Sampled events over sub-stream i in stream w
$N_{i,t}^w$	Number of events observed in sub-stream i in stream w at time t
θ_i^w	Sample rate defined by Aion over sub-stream i in stream w
ddl_i^w	Deadline for the ith sub-stream in stream w

Table 1: Symbols Used

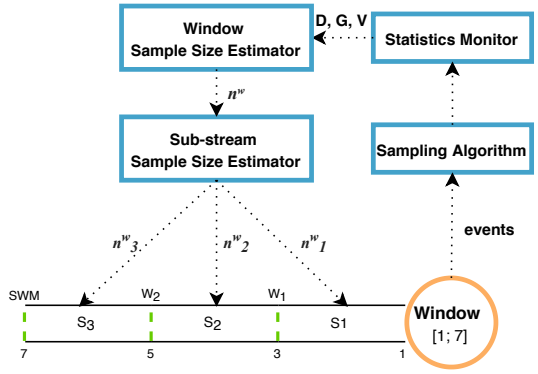


Figure 5: Example illustrating Aion components' interaction over a logically divided stream.

Aion is composed of (i) an algorithm that monitors properties of the workload including the network delay, the inter-event generation delay, and the distribution of the event values (Section 3.1), (ii) a proactive algorithm that estimates the sample size such that the error margin is bounded by r_{thr} (Sections 3.2 and 3.3), and (iii) a sampling algorithm that effectively samples the input based on the computed sample size (Section 3.3).

To illustrate Aion's functionality, consider Fig. 5 of a window operator encompassing all events generated between 1 and 7, with watermark frequency of $f = 2$. Initially, the Statistics Monitor collects and stores information such as the network delay (D), inter-event generation delay (G), and the event values (V). Then, the Window Sample Size Estimator is invoked either to build an estimation on the targeted windowed sample size or to update an existing one (Sec. 3.2). The sample size is estimated based on the specified error margin r_{thr} and the type of the windowed operator. For a window

stream w, this part of the algorithm outputs n^w as the desired sample size estimate for the entire windowed stream. Finally, n^w is forwarded to the Sub-Stream Sample Size Estimator that computes the desired sample size for each sub-stream. Aion computes the sample size as a function of n^w and the expected ingestion delay of stragglers. Then, at each sub-stream, Aion runs its sampling algorithm to choose a sample with the target sampling size that is fair and representative.

Each of the aforementioned components in Fig. 5 are described in detail in the following sections.

3.1 Monitoring the Workload

Aion collects necessary information from the stream to quantify the target sampling size that achieves the output accuracy guarantees $P(r \leq r_{thr}) \geq 1 - \delta$ (Fig. 5). In this section, we discuss in detail the collected information, their statistical representation, and their role in Aion.

Aion collects its information on a sub-stream basis to ascertain high output accuracy and to leverage incremental window processing. For the i^{th} sub-stream in the windowed stream w, Aion collects from the stream three main pieces of information: the network delay d_i^w, the inter-event generation delay g_i^w, and the event values distribution v_i^w. We denote by $d_i^w = \{d_{i,0}^w, d_{i,1}^w, ...\}$ the set of observed network delays by the SPE over the i^{th} sub-stream. For an event e, the network delay can be computed by $e.rts - e.gts$, where $e.rts$ refers to e's ingestion time by the SPE, and $e.gts$ to its generation time at the source. Similarly, for every two consecutive events e_k and e_{k+1}, $g_i^w = \{g_{i,0}^w, g_{i,1}^w, ...\}$ encompasses the set of all inter-event generation delays computed by $e_{k+1}.gts - e_k.gts$. Finally, $v_i^w = \{v_{i,0}^w, v_{i,1}^w, ...\}$ contains the set of event values observed in the corresponding sub-stream i in stream w. In Aion, this information is collected at the ingestion of every new event (Algorithm 1). More specifically, after identifying the windowed stream and the sub-stream to which event e belongs based on its generation timestamp (lines 3–4), the necessary information is then extracted (lines 6–10).

Aion utilizes the collected information over the processed sub-streams to proactively estimate the patterns of the upcoming sub-streams. For a processed sub-stream i whose watermark has been emitted, we capture the statistical significance of the collected information distribution by the mean and the standard deviation. More specifically, we denote mean network delay by $\mu(d_i^w) = \frac{1}{|d_i^w|}\sum_{j=0}^{|d_i^w|} d_{i,j}^w$ and $\sigma(d_i^w)$ as the standard deviation. Similarly, We define $\mu(g_i^w)$, $\sigma(g_i^w)$, and $\mu(v_i^w)$, $\sigma(v_i^w)$, for inter-event generation delays and event values distribution to follow the above definitions. Aion computes the mean delay and the standard deviation on-the-fly, imposing no storage or computational overhead. Furthermore, Aion does not assume any underlying distribution over the collected information as information patterns can vary over multiple distributions [38] As for an upcoming sub-stream i, Aion estimates the statistical representation of the needed information i.e., d^w, v^w, g^w based on the historically processed sub-streams. We denote by the random variables D_i^w, G_i^w, and V_i^w the network delay, the inter-event generation delay, and the event values, respectively. Then, for the upcoming sub-stream i, we estimate the mean for D_i^w by:

$$E[D_i^w] = \frac{1}{m}\sum_{j=i-1-m}^{i-1} E[D_j^w] = \frac{1}{m}\sum_{j=i-1-m}^{i-1} \mu(d_j^w) = \mu(d_i^w) \quad (1)$$

Note that our estimations are limited to the last m sub-streams to reduce the storage overhead. .

Since D_i^w is the result of a summation of means, D_i^w follows a normal distribution through the central limit theorem. Having known distributions, specifically normal distribution, provides reliability on calculations using the aforementioned random variables. These reliability properties are also shared by G_i^w and V_i^w since we define them similarly to Eq. 1.

Aion leverages these random variables for key calculations. That is, in Algorithm 2, Aion utilizes the network delay and the inter-event generation delay to collect the sub-stream size (line 4), and it utilizes the event values to estimate the sample size in Algorithm 1 (line 16) and Algorithm 2 (line 10). We discuss the estimations further in the next section.

Algorithm 1 Aion: Processing Events

```
1:  procedure processEvent(e)
2:      /* Identify the stream and sub-stream event i belongs to */
3:      w ← getWindowIndex(e)
4:      i ← getSubStreamIndex(e)
5:      /* Examples of the collected statistics */
6:      d_i^w ← d_i^w ∪ (e.rts − e.gts)
7:      g_i^w ← g_i^w ∪ e.gts
8:      v_i^w ← v_i^w ∪ e.val
9:      w.observedEvents ← w.observedEvents + 1
10:     i.observedEvents ← i.observedEvents + 1
11:     /* If watermark was already emitted for ith substream */
12:     if i.isProcessed then
13:         dropEvent(e)
14:         return
15:     end if
16:     n_i^w ← GetSampleSize(w, i)
17:     s_i^w ← getSubsample(w, i)
18:     /* Sampling Alg chooses whether to sample e or drop it */
19:     isSampled ← runSamplingAlgorithm(s_i^w, θ_i^w)
20:     if !isSampled then
21:         dropEvent(e)
22:     else
23:         s_i^w ← s_i^w ∪ e
24:     end if
25:     /* Check if requirements are met to generate a watermark */
26:     if SafeToProcess(n_i^w, s_i^w) then
27:         genWatermark(w, i)
28:         processSample(s_i^w)
29:         i.isProcessed ← True
30:     end if
31: end procedure
32: procedure SafeToProcess(n_i^w, s_i^w)
33:     minNeededSize ← getTargetSize(w, i)
34:     return currTime ≥ i.ddl && n_i^w ≥ minNeededSize
35: end procedure
```

3.2 Window and Sample Size Estimators

Aion intelligently selects and processes a sample which delivers a result within r_{thr} of the true result. Initially, Aion quantifies the projected number of events in the windowed stream. Then, based on the estimated number of events, r_{thr}, and the type of the window operator, Aion estimates a target sample size. This section discusses Aion's estimation techniques for the window and sample sizes.

Initially, Aion quantifies the projected number of events in the windowed stream based on the collected information (Section 3.1). For a windowed stream w, we denote the size of the window by N^w representing the total number of events including stragglers of the window. Intuitively, the size of the windowed stream is a function of the size of each of its sub-stream constituents, that is, $N^w = \sum_{i=1}^{k} N_i^w$, where k represents the number of sub-streams in the windowed stream w, and N_i^w refers to the number of events in sub-stream i (Algorithm 2, lines 3–6). The size for each sub-stream is then estimated based on the inter-event generation delays observed over previously processed sub-streams. Note that a workload with a high frequency of event generation would entail low values of g_i^w, while sparser event generation would mean higher values of g_i^w. The size estimation of sub-stream i can be expressed by:

$$E[N_i^w] = \frac{f}{E[g_i^w]} = \frac{f}{\mu(g_i^w)} \tag{2}$$

Aion continuously adjusts its estimations of the size of the windowed stream and its sub-stream constituents as earlier sub-streams are processed (i.e., as corresponding watermarks are emitted). In doing so, it guarantees more accurate estimations as time progresses towards the window's deadline. It is important to note that regardless of the distribution of g_i^w which can vary depending on the application type [36], Aion makes no assumptions on the input's arrival rate.

Aion computes the sample size based on the statistics monitor's estimation of N^w. However, since estimation elicits uncertainty thereby affecting the accuracy of the sample size, Aion seeks to overestimate N^w based on the level of uncertainty quantified by the standard deviation. In doing so, the sample size is marginally augmented to achieve higher accuracy guarantees while keeping it small enough to maintain low processing cost. Overestimation of N^w is extremely helpful in the case of ingesting more events than the anticipated size. As for underestimation, the processing cost is marginally increased, thereby hardly affecting it. Hence, marginally overestimating N^w helps Aion to consistently achieve robust performance in the face of workload fluctuation. In our experiments, we overestimate the window length based on two standard deviations.

After estimating the windowed stream size N^w, we quantify the desired sample size (Algorithm 2, line 8) based on r_{thr}, and the type of the window operator. From the related literature, there has been work on sample processing that did not account for the functionality of the window operator and therefore limited their sample selection to achieving similar statistical properties to that of the original input [25]. However, as illustrated and recommended in [11], specifying an error function based on the window functionality yields consistent higher accuracy. Aion adopts the latter approach to achieve the highest accuracy possible. We provide two examples

Algorithm 2 Aion: Estimations & Sampling

```
1:  procedure getSampleSize(w, i)
2:      /* Estimate the window size for each sub-stream (Sec. 3.2) */
3:      for j in range (0, (ddl^w − ddl^{w−1}) / f) do
4:          N_j^w ← estmSubstreamSize(w, j) // (Eq. 2)
5:          N^w ← N^w + N_j^w
6:      end for
7:      /* Estimate the sample size for each sub-stream (Sec. 3.3) */
8:      n^w ← estmWindowSampleSize(N^w) // (Eq. 4
9:      for j in range (0, (ddl^w − ddl^{w−1}) / f) do
10:         n_j^w ← estmSubstreamSampleSize(n^w, j) // (Eq. 8)
11:     end for
12:     return n_i^w
13: end procedure
14: procedure runSamplingAlgorithm(n_i^w, e)
15:     θ_i^w ← getSamplingRate(n_i^w) // (Eq. 8)
16:     r ← genNumber(0, 1)
17:     return r ≤ θ_i^w
18: end procedure
```

of error function derivations for the two commonly used window functionalities: events-mean computation, and summation. For each functionality, we derive a formula relating the error bound r_{thr}, the windowed stream size N^w, and its corresponding sample size n^w.

We consider the first case of events-mean computation, where the window is computing the average of observed events. The relative error function can be expressed as $\frac{|\mu(s^w)-\mu(v^w)|}{\mu(v^w)}$, where $\mu(s^w)$ represents the mean of the sampled input, and $\mu(v^w)$ denotes the mean of the original input. To maintain the processing cost at a minimum, we are interested in finding the *minimum* sample size n^w that satisfies $P(|\frac{\mu(s^w)-\mu(v^w)}{\mu(s^w)}| \leq r_{thr}) \geq 1-\delta$. Per [30], the error r_{thr} is tightly related to sample size n^w, and the original input size N^w by:

$$r_{thr} = \frac{z_{\delta/2}\sqrt{(1-\frac{n^w}{N^w})} \times \frac{\sigma(s^w)}{\sqrt{n^w}}}{\mu(v^{w-1})} \tag{3}$$

where $z_{\delta/2}$ refers to the confidence interval matching the z-valuewith the specified probability δ. Then, solving for n^w, we have:

$$n^w = \frac{z_{\delta/2}^2 \sigma^2(s^w)}{r_{thr}^2 \mu(v^{w-1})^2 + \frac{z_{\delta/2}^2 \sigma(s^w)^2}{N^w}} \tag{4}$$

By solving for the minimum sample size in Eq. 3, n^w can be derived as in Eq. 4. Thus, Aion collects at least n^w events in the windowed stream w to achieve the accuracy guarantees. The $\mu(v^{w-1})$ is a historical mean.

n^w/N^w is the proportion of events that were sampled from the original input. Therefore, taking the estimate for the sample total and scaling it up by N^w/n^w accounts for events that are not in the sample. Using the derived equations for events-mean computation, a formula for estimating the input summation is given by:

$$\frac{N^w}{n^w}\sum_{i=1}^{n^w} s_i^w = N^w \frac{\sum_{i=1}^{n^w} s_i^w}{n_w} = N^w \mu(s^w) \quad (5)$$

Eq. 5 expresses that an estimate for the original input can be taken by scaling $\mu(s^w)$ up by the known N^w. This method requires finding an estimate for $\mu(s^w)$ and therefore, the same sample size estimate for determining $\mu(s^w)$ as seen in Eq. 4 can be used to get an accurate estimate for the total input summation.

As for the summation operator, that is, computing the sum over all elements considered, the relative error function is defined by:

$$r = \frac{|\sum_{i=1}^{N^w} v_i^w - (N^w \times \mu(s^w))|}{|\sum_{i=1}^{N^w} v_i^w|} \quad (6)$$

There exist multiple approaches to define error bounds over a window functionality. For instance, a different error function for the summation functionality is used by AQ-K-Slack [24]. This approach imposes an unnecessarily large sample size. The literature also includes work on other types of scalar window functions like MAX, MIN, and quantiles [12, 26]. Other event-based vector operations can be adapted for Aion by utilizing the error functions derived as in [3, 26] for grouping. They can be incorporated into Aion following similar derivations from Sec. 3.2.

After estimating the window and sample sizes, Aion utilizes these estimations in its "Sub-stream Sample Size Estimator" component (Fig. 5). Aion leverages these estimations over each sub-stream and then executes its sampling algorithm accordingly, as described in the next section.

3.3 Sampling over Sub-Streams

Aion is optimized to select a sample free of stragglers that is representative of the original input. In this section, we describe Aion's sampling algorithm that minimizes the impact of stragglers.

On the arrival of each event, Aion runs its sampling algorithm (line 16 in Algorithm 1, procedure call defined in Algorithm 2 line 14) to select its sample. Based on the computed sample size n^w (Section 3.1, and line 10 in Algorithm 2), an event can be added to the sample based on the rate $\frac{n^w}{N^w}$. However, this approach suffers from the stragglers' problem as the sample is unlikely to be complete by the window's deadline. Aion, therefore, optimizes the sampling rate over each sub-stream i so that at the window's deadline, the sample would be complete or near completion. To express this formally, we denote $N_{i,t}^w$ to be the total number of events ingested by time t. By using the inter-event generation and network delays, we can estimate $N_{i,t}^w$ by:

$$N_{i,t=ddl_i^w}^w = \frac{(ddl_i^w - E[D_i^w]) - ddl_{i-1}^w}{E[G_i^w]} \quad (7)$$

Then, the updated sampling rate at sub-stream i is computed by:

$$\theta_i^w = \frac{n^w}{N_{i,t=ddl_i^w}^w} \quad (8)$$

Aion then samples the incoming events according to θ_i^w thereby mitigating the presence of stragglers. Note that if θ_i^w is greater than 1, Aion includes the minimum number of stragglers to meet n thereby imposing minimal slack delay. The algorithm is shown in Algorithm 2 (lines 14–17).

As such, Aion minimizes the numbers of stragglers in the sample by prioritizing sample completion before the window's deadline. Then, Aion generates a watermark as soon as the window's deadline is due and the accuracy guarantees are achieved (Algorithm 1, lines 32–34). These conditions ensure that a watermark is generated when the accuracy guarantees are met while minimizing the output latency.

4 PERFORMANCE EVALUATION

In this section, we present the results of a series of experiments conducted on multiple benchmark workloads to demonstrate the performance advantage that Aion possesses over representative algorithms from prior work.

4.1 Experimental Setup

We describe our experimental setup and methodology, including machine configurations, the sampling algorithms that we compare Aion against, benchmarks, and the delay distribution settings. Our experiments are run on a machine having an Intel Xeon processor consisting of 24 cores (using hyper-threading) and 32 GB of memory. The machine is running Java OpenJDK implementation v1.8.0_191 on top of Ubuntu 16.04 LTS. The implementation of Aion is on Apache Flink v1.8. We dedicate a different machine with the same configuration for generating the workload, which is transmitted to the SPE nodes via Kafka v2.2.1.

4.1.1 Benchmarks. We conduct our evaluation using three well-known streaming benchmarks: the Yahoo! Streaming Benchmark (**YSB**) [20], the New York City Taxi (**NYT**) [23], and the **kMeans** benchmark. We implement these benchmarks on Apache Flink and evaluate performance by running different sampling algorithms in each experiment. YSB emulates an advertisement tracking system where users launch ad campaigns, each of which is composed of multiple ads. The YSB query handles a stream of ad clicks and outputs the interest in each ad campaign. We use the code-base provided by [20] with the addition of generating periodic watermarks from the source. NYT covers a large dataset of taxi trips in New York, spanning six years. The dataset is rich with information such as the number of passengers, distances, and fares. The query measures the average distance of each trip ride in sliding windows. KMeans query is an algorithm that partitions the dataset into k clusters. The dataset originally at the source is filtered and processed in the pipeline before running the kMeans algorithm in a windowed operator.

4.1.2 Algorithms. Sample processing algorithms generally need to run in conjunction with a deployed stream progress control algorithm. Specifically, sample processing algorithms continuously update their sample until the accompanying stream progress control algorithm generates a watermark. Popularly used stream progress control algorithms include slacking techniques [33, 36]. Slack algorithms compute the minimum slack delay needed to guarantee input completion, then generating a watermark as soon as the slack delay expires. We implemented onto our system the K-Slack [33]

algorithm to automate the generation of watermarks. K-Slack generates a watermark every k seconds, where k is set to the maximum observed network delay. Since Aion uniquely combines sampling and stream progress control, it overrides K-Slack by running its own stream progress control algorithm proposed in this paper.

To demonstrate Aion's efficacy, we compare Aion against *Default*, which is the baseline approach that constitutes running Flink with no sampling. In Default, windows process their input as soon as the deployed stream progress control algorithm emits a watermark. We also compare against the following sample processing algorithms:

- AQ-K-Slack [24]: Similarly to Aion, this algorithm samples events as they are ingested by the SPE. It computes the sampling ratio needed to achieve the necessary output accuracy guarantees. Using an error function specific to summation windowed functionality, AQ-K-Slack ties the relative error function with the target accuracy while accounting for stragglers in processing output.
- Aion-: denotes Aion without (minus) the stragglers' circumvention technique employed (Sec 3.3). The Aion- algorithm applies Aion's sampling technique but without the stream progress control algorithm. Specifically, the sample size is estimated as a fraction of N_i^w and not $N_{i,t=ddl_i^w}^w$. We implemented Aion- to study the impact of circumventing stragglers when sampling.

4.1.3 Delay Distributions. As in [36] , we vary two main types of delay for our experiments, namely the network delay, and the inter-event generation delay. Similarly to [36], we refer to these as follows in our experiments;

- CC: Network and inter-event generation delays are constant at 150ms and 1ms, respectively.
- GG: Network delay and inter-event generation delay are distributed with $Gamma(k = 60, \theta = 4)$ and $Gamma(k = 2, \theta = 0.5)$ respectively.
- EC: Network delay is exponentially distributed with mean delay of 240ms while inter-event generation delay is constant at 1ms.
- EG: Network delay is exponentially distributed with mean delay of 240ms while inter-event generation delay is Gamma distributed with $Gamma(k = 2, \theta = 0.5)$.

We generate ten different streams for each combination. Each data point on the graph is an average over at least 10 independent runs (unless stated otherwise). Ideally, Aion's watermark periodicity f should be a divisor of the window deadline so that a watermark can be emitted at the window's deadline, effectively minimizing latency. Based on empirical evidence collected through our experiments, we set f to 600ms for deployed windows of size 3s and 6s. This value strikes a balance between the estimation granularity and the overhead of the algorithm.

4.2 Results

4.2.1 YSB. Figure 6 shows the performance of the sampling algorithms running the YSB workload under different delay distributions. For the first distribution of UU, Aion delivers lower latency than the other algorithms as it reduces both the stragglers' impact

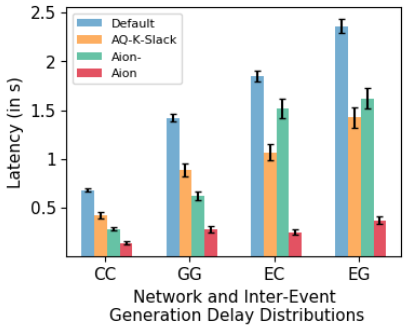
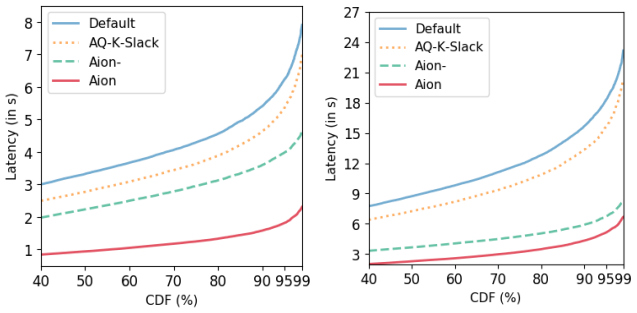


Figure 6: Mean latency vs. different environment distributions of network delay and inter-event generation delay running YSB benchmark.

on output latency and the processing cost of events. More specifically, while Default, AQ-K-Slack, and Aion- algorithms impose 150ms delay to account for stragglers, Aion circumvents the slack delay by emitting a watermark as soon as the output accuracy guarantees are achieved. For the processing cost, since Default processes all of the events, its processing overhead dominates the output latency. As for Aion- and Aion, both algorithms sampled at a rate reaching 30% of the original input, while the AQ-K-Slack sampling mechanism is more restrictive, pushing its sampling rate to exceed 60%. Aion minimizes both the slack delay and the processing cost yielding significant latency reduction over the other algorithms. For the GG delay distribution, the maximum observed network delay reached 350ms adding significant overhead for both Default and AQ-K-Slack. However, as in the case of UU, the processing cost dominated the output latency for both Default and AQ-K-Slack. Aion delivers lower output latency over Default and AQ-K-Slack by 80% and 70%, respectively.

When network delay is exponentially distributed, the performance of Default and Aion- worsens as the maximum observed network delay exceeds 1500ms. That is, the two algorithms process windowed data long after generating a watermark that is past the window's deadline. Note that Aion- quickly processes its data after slacking for 1500ms while Default processes the entire input. As for AQ-K-Slack, it delivers better performance over these two algorithms since its slack function uses randomization to mitigate the delay. Since AQ-K-Slack's sampling function is costly, its processing cost dominates output latency. Aion outperforms Default and AQ-K-Slack with latency reductions of 85% and 78%, respectively. Aion significantly outperforms the two algorithms by mitigating the effect of stragglers and significantly reducing the processing cost.

We also compared Aion's latency to the other algorithms for different input loads. Fig. 7 presents the cumulative distribution function (CDF) of recorded latencies for the range of 40^{th} to 99^{th} percentile tail latency under two input load levels of number of events generated: 5, 000 and 25, 000 (5x) events/s. The experiment



(a) 5,000 **events/s** **(b)** 10,000 **events/s**

Figure 7: CDF latency running YSB benchmark for delay distribution EG with 5,000 **and** 25,000 **number of input events generated per second.**

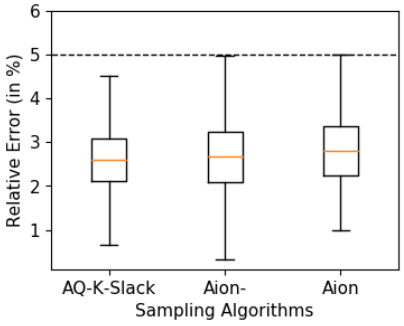


Figure 8: Error plot showing statistical significance obtained by AQ-K-Slack, Aion-, and Aion running YSB benchmark for delay distribution EG with 25,000 **input events generated per second.**

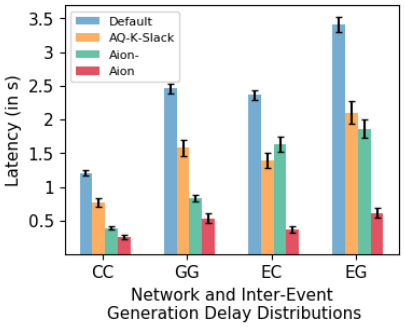


Figure 9: Mean latency vs. different environment distributions of network delay and inter-event generation delay running NYT benchmark.

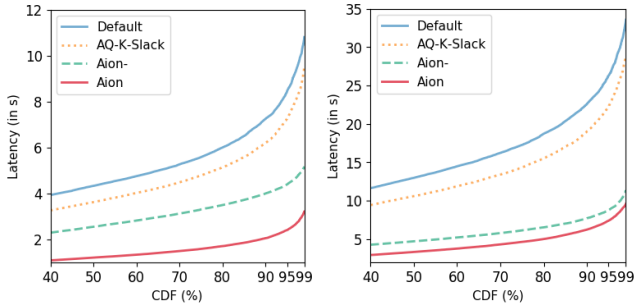
is run with distribution EG where the network delay is exponentially distributed and the inter-event generation delay is gamma-distributed. For the 90^{th} percentile, Aion achieves latency reductions over Default and AQ-K-Slack of 76% and 70% respectively for both 5,000 and 25,000 events generated per second. As for tail latencies, and specifically 95^{th} to 99^{th} percentiles, Aion maintains a low latency level delivering consistent latency performance as with smaller percentiles. However, this was not the case for other algorithms as Aion reduced latency over Default and AQ-K-Slack by 80% and 75% respectively for 5,000 events generated per second, and 84% and 80% respectively for 25,000 events generated per second. Interestingly, Aion- performed similarly to Aion for 25,000 events/s but not for 5,000 events/s. This is due to the output latency for 25,000 events/s being dominated by the processing cost and not the slack cost, while for 5,000 events/s it was dominated by the slack delay. Aion optimizes for both of these costs to attain the large aforementioned latency reductions.

Aion optimizes for reduced latency while still achieving the specified accuracy guarantees. Figs. 6 and 7 show that Aion outperforms the other algorithms significantly in terms of reducing output latency. Next, we show the accuracy guarantees delivered by the sampling algorithms Aion, Aion-, and AQ-K-Slack through experiments with parameters $r_{thr} = 5\%$ and $\delta = 0.95$.

Fig. 8 presents the distribution of error obtained while running YSB. The experiments were executed for EG with 25,000 events generated per second. For the three tested algorithms, although the relative error margin was set to 5%, the algorithms recorded significantly lower error margins. Specifically, the mean relative error (Sec. 3.2) reached 3% for all algorithms with a relatively small standard deviation. Interestingly, for all algorithms, the maximum experienced error was less than the threshold, thereby delivering excellent accuracy exceeding the required guarantees. As such, this figure proves that circumventing stragglers have little to no impact on finding a sample with high accuracy guarantees.

4.2.2 NYT. The NYT benchmark is a relatively expensive query compared to YSB as it includes a longer pipeline and more processing intensive operators. As such, the processing cost in NYT factors higher in output latency compared to the processing cost factor in YSB's output latency. Fig. 9 shows the performance of the sampling algorithms running the NYT workload under different delay distributions. For UU, Aion achieves significantly lower output latency over Default and AQ-K-Slack mainly due to Aion's superior sampling mechanism as it samples at a lower rate. Since the slack delay in UU is minimal and processing cost of NYT is expensive, Aion achieves a latency reduction of 75% over Default that processes all events, and 60& reduction over AQ-K-Slack as it suffers from higher processing cost and a high slack delay. For other delay distributions, and specifically EU and EG, where the experienced network delay is significantly higher, Aion's mechanism of circumventing stragglers significantly reduced the output latency. Specifically, Aion achieved an 80% output latency reduction over Default and 72% over AQ-K-Slack. Furthermore, the difference between Aion and Aion- in EU and EG indicate that it is essential to minimize the straggler count in the sample to minimize the output latency, regardless of the query complexity.

Aion's latency CDF for NYT is shown in Fig. 10. As explained earlier, this experiment delivered different results than that for YSB in Fig. 7 due to the higher complexity of NYT benchmark. At the lighter workload of 5,000 events per second, the output latency of Aion- was dominated by slack delay while Default and AQ-K-Slack



(a) 5,000 **events/s** **(b)** 10,000 **events/s**

Figure 10: CDF latency running NYT benchmark for delay distribution EG with 5,000 **and** 25,000 **number of input events generated per second.**

suffered from both the slack delay and processing cost. Consequently, Aion's tail latency, 95^{th} to 99^{th} percentile, is significantly reduced over Default and AQ-K-Slack by 80% and 75%, respectively. As for the heavier workload of 25,000 events per second, the tail latency reduction attained by Aion over Default and AQ-K-Slack reaches 87% and 85%, respectively. Aion- performs similarly to Aion as output latency is dominated by the processing cost in both of these Aion-based algorithms. Aion incurs lower processing cost because its sampling mechanism requires a significantly smaller sample size. Therefore, Aion can attain greater scalability than other algorithms as it avoids slack delays while reducing processing costs.

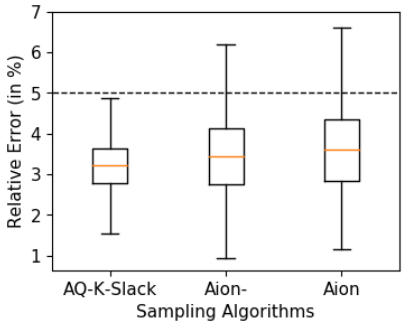


Figure 11: Error plot showing statistical significance obtained by AQ-K-Slack, Aion-, and Aion running NYT benchmark for delay distribution EG with 25,000 **input events generated per second.**

We studied the distribution of errors obtained when running the NYT benchmark in Fig. 11 with parameters $r_{thr} = 5\%$ and $\delta = 0.95$. In this case, since the NYT window operator is a mean estimation and the workload entails a higher standard deviation of V, both Aion- and Aion experience error greater than 5%. However, the error crossed the threshold only three times from one-hundred data points collected, therefore achieving a high confidence level of 97%. Furthermore, while AQ-K-Slack delivered a significantly higher latency, by 80%, the error for both algorithms was very similar. Finally, Aion's stringent accuracy requirements are a function of δ as Aion guarantees $P(r \leq r_{thr}) \geq \delta$. But in the cases where the obtained error exceeded r_{thr}, the max error is relatively close to r_{thr} indicating that Aion delivers consistent accuracy performance.

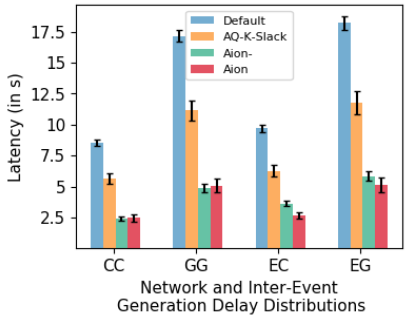
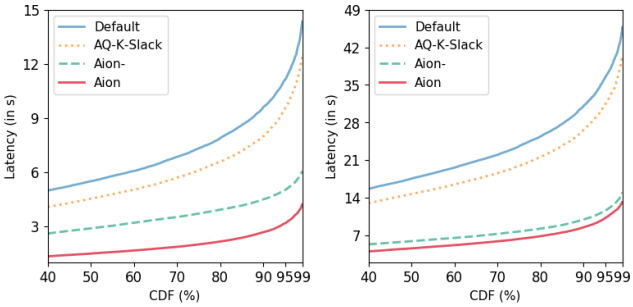


Figure 12: Mean latency vs. different distributions of network delay and inter-event generation delay running kMeans benchmark.

4.2.3 kMeans. The benchmark kMeans is the third and most expensive query for which we compare Aion against the other algorithms. Furthermore, the sampling rate in kMeans reached 45% for the Aion algorithms, whereas it reached 85% for AQ-K-Slack. In this case, the processing cost dominates the output latency for all tested configurations. This is shown in Fig. 12 as the two Aion algorithms delivered significantly lower output latency results over Default and AQ-K-Slack for all environment settings. Aion's sampling mechanism helped it to reduce the output latency significantly. Aion minimized both the processing cost and the network penalty over Default and AQ-K-Slack by up to 75% and 65%, respectively.



(a) 5,000 **events/s** **(b)** 10,000 **events/s**

Figure 13: CDF latency running kMeans benchmark for delay distribution EG with 5,000 **and** 25,000 **number of input events generated per second.**

Aion's latency CDF for kMeans is shown in Fig. 13. The benchmark kMeans is even more expensive than NYT, making processing cost the dominating factor in output latency. For 5,000 events generated per second, Aion achieves significantly lower 95^{th} to 99^{th} percentile tail latencies, reducing them over Default and AQ-K-Slack by about 80% and 72%, respectively. AQ-K-Slack, which

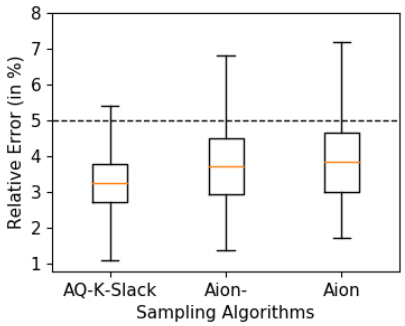


Figure 14: Error plot showing statistical significance obtained by AQ-K-Slack, Aion-, and Aion running kMeans benchmark for delay distribution EG with $25,000$ input events generated per second.

suffers from a significantly higher sampling rate, incurs high output latency thereby displaying similar tail latency growth to that of Default. As for the $25,000$ events generated per second input load, we observe that Aion and Aion- achieve very similar performance, further indicating that the processing cost dominated the output latencies in this experiment. As for Default and AQ-K-Slack, both algorithms suffered significant performance degradation, adding 82% and 78% latency overhead, respectively, at the tail. Aion delivers slightly better performance over Aion- since Aion exhibits no slack delay while Aion- waited for an additional 1.5s to ensure input completion.

Similar to previous experiments, we ran experiments with the following parameters: $r_{thr} = 5\%$, $\delta = 0.05$. Fig. 14 presents the distribution of error obtained while running kMeans. In this case, since the window operator is computing the k-means algorithm that involves more complex processing, the resulting error was higher. While all algorithms achieved a maximum latency higher than that of the threshold, all of them achieved this within the constraint $P(r \leq r_{thr}) \geq 1 - \delta$. Furthermore, while AQ-K-Slack delivered significantly higher latency by 80%, the mean error for both algorithms was very similar. Empirical verification and experimentation consistently demonstrated that Aion error values rarely stretched over the specified error threshold.

We measured Aion's runtime overhead incurred by statistics collection, window and sample size estimation, and sampling incoming events. Aion's overhead is about 0.15% of the measured output latency per window. This low overhead is due to Aion's minimal monitoring of workload and its efficient sampling technique.

5 RELATED WORK

Approximate Query Processing (AQP) has been applied in multiple systems. For relational databases, AQP has been studied extensively [2, 11, 29] with techniques based on sampling [4], sketches [18], and aggregations [22]. These techniques try to expedite query processing latency and memory utilization by approximating the results of the query. Such AQP techniques inherently assume that (i) databases have their data stored on disk and can, therefore, sample offline [10], and (ii) read-heavy databases can be exploited to build and store high-quality samples [29]. However, these assumptions do not apply to SPEs as events are ingested continuously, are not stored, and are then processed on the fly.

AQP has also been studied in the context of SPEs [14, 25]. Sketches has been applied in the context of reducing memory footprint of stateful operators [9, 13, 16, 37]. Sketches utilize various data structures to store statistically significant information about the original input. However, sketches are designed to reduce memory utilization by running algorithms to summarize the input. As such, these techniques that are geared towards reducing memory utilization cannot be efficiently adopted to reduce output latency at the expense of accuracy.

Earlier AQP research proposed a framework to implement sample processing algorithms on the Gigascope SPE [25]. Examples of common sample processing approaches discussed in [25] include [24, 31, 35, 40, 41] that optimize for different goals in the system. For instance, [6] gracefully degrades performance by dropping events when the system is overloaded. This proposal computes the minimum shedding rate such that system resources are not over-utilized while still maintaining adequate query accuracy. A proactive load-aware shedding (LAS) mechanism that aims to limit queue latencies was proposed [35]. LAS utilizes learning techniques thereby advancing a model-free algorithm to assess the cost of processing events. Another approach is presented in [24] whereby the sample size is estimated for aggregations with dynamic slack time for window processing. However, these methods generally wait for the arrival of watermarks before processing the sample. As previously illustrated, methods that include stragglers for their input are prone to suffer from high output latency.

Sample processing has also been studied in different contexts. IncApprox [27] is an approximation method that studies the problem of having the deployed SPE ingest events from multiple sources, each of which is emitting at a different rate. IncApprox presents a solution that represents the data received from each source as *strata* through stratified sampling. The weight of each stream is then computed based on the arrival rate. Reservoir sampling is then applied to each source which is used for approximation. However, IncApprox is designed for systems relying on emitting mini-batches as in Apache Spark [45]. StreamApprox [34], an evolution of IncApprox, relaxes this assumption by presenting a system designed for both mini-batches and event-at-a-time processing paradigms. While Aion does not discriminate between stream sources, the algorithm can be easily extended by stratifying each stream source as described in StreamApprox and IncApprox. These papers assume that the user is knowledgeable enough to provide a static sample size that minimizes the error rate, raising practicality concerns. Additionally, all of these algorithms suffer from the stragglers problem.

There exists a multitude of techniques proposed to include stragglers by imposing extra latency to ensure input completion [14, 17, 24, 33, 36, 38, 44]. Heartbeats [38] are watermark-like events injected in the stream to signify end of sub-streams. This algorithm assumes static maximum latency delay and consequently is prone to suffer from high latency. Similar work [33] proposed a watermark generation algorithm that processes an event only if has been idle in the stream for k seconds where k is set to the maximum

observed network delay. More recently, [36] proposed an algorithm with probability guarantees on slacking for stragglers by monitoring both network and inter-event generation delays. However, these algorithms present a solution against the uncertainty of the number of stragglers by imposing slack delay. In contrast, Aion's novelty lies in its observation that stragglers are unproductive to sample processing, thereby circumventing them and minimizing slack delay to deliver minimized output latency.

6 CONCLUSION

In this paper, we presented Aion, a state-of-the-art sample processing algorithm optimized to reduce output latency while achieving specified accuracy guarantees. Aion mitigates the impact of stragglers on output latency by leveraging control on stream progress, effectively automating the generation of watermarks. Through integration into Apache Flink and extensive experimentation, we demonstrated that Aion outperforms existing sampling techniques. Our experiments show that Aion can deliver large output latency reductions of up to 85% over existing techniques. These results demonstrate the effectiveness of Aion's intelligent sampling design for circumventing stragglers in delivering superior stream processing performance with accuracy guarantees.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation and Ontario Research Fund.

REFERENCES

[1] Zainab Abbas, , et al. 2018. Streaming graph partitioning: an experimental study. In *PVLDB*, Vol. 11. 1590–1603.
[2] Swarup Acharya et al. 1999. The Aqua approximate query answering system. In *SIGMOD*. ACM, 574–576.
[3] Swarup Acharya et al. 2000. Congressional samples for approximate answering of group-by queries. In *SIGMOD*. ACM, 487–498.
[4] Sameer Agarwal et al. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EUROSYS*. ACM, 29–42.
[5] Tyler Akidau et al. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *PVLDB*, Vol. 6. ACM, 1033–1044.
[6] Brian Babcock et al. 2004. Load shedding for aggregation queries over data streams. In *ICDE*. IEEE, 350–361.
[7] Magdalena Balazinska et al. 2007. Moirae: History-Enhanced Monitoring.. In *CIDR*. 375–386.
[8] Paris Carbone et al. 2015. Apache flink: Stream and batch processing in a single engine. In *TCDE*, Vol. 36. IEEE, 28–38.
[9] Moses Charikar et al. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.
[10] Surajit Chaudhuri et al. 2007. Optimized stratified sampling for approximate query processing. In *TODS*, Vol. 32. ACM, 9–es.
[11] Surajit Chaudhuri et al. 2017. Approximate query processing: No silver bullet. In *SIGMOD*. ACM, 511–519.
[12] Graham Cormode et al. 2006. Space-and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS*. ACM, 263–272.
[13] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. In *Journal of Algorithms*. Elsevier, 58–75.
[14] Miyuru Dayarathna and Srinath Perera. 2018. Recent advancements in event processing. *ACM Computing Surveys (CSUR)*, 1–36.
[15] Alan Demers et al. 2006. Towards Expressive Publish/Subscribe Systems. In *EDBT*. Springer, 627–644.
[16] Edward Gan et al. 2018. Moment-Based Quantile Sketches for Efficient High Cardinality Aggregation Queries. In *PVLDB*, Vol. 11. ACM, 1647–1660.
[17] Nikos Giatrakos et al. 2020. Complex event recognition in the big data era: a survey. In *VLDBJ*. Springer, 313–352.
[18] Phillip B. Gibbons and Yossi Matias. 1998. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *SIGMOD*. ACM, 331–342.
[19] Lukasz Golab and M Tamer Özsu. 2003. Issues in data stream management. In *SIGMOD*. ACM, 5–14.
[20] Jamie Grier. 2016. Extending the yahoo! streaming benchmark. *URL http://data-artisans.com/extending-the-yahoo-streamingbenchmark* (2016).
[21] Michael Grossniklaus et al. 2016. Frames: data-driven windows. In *DEBS*. ACM, 13–24.
[22] Joseph M Hellerstein et al. 1997. Online aggregation. In *SIGMOD*. ACM, 171–182.
[23] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 Grand Challenge. In *DEBS*. ACM, 289–294.
[24] Yuanzhen Ji et al. 2015. Quality-Driven Continuous Query Execution over Out-of-Order Data Streams. In *SIGMOD*. ACM, 889–894.
[25] Theodore Johnson et al. 2005. Sampling algorithms in a stream operator. In *SIGMOD*. ACM, 1–12.
[26] Nikos R Katsipoulakis et al. 2020. SPEAr: Expediting Stream Processing with Accuracy Guarantees. In *ICDE*. IEEE.
[27] Dhanya R Krishnan et al. 2016. Incapprox: A data analytics system for incremental approximate computing. In *WWW*. 1133–1144.
[28] Jin Li et al. 2008. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. In *PVLDB*, Vol. 1. ACM, 274–288.
[29] Kaiyu Li and Guoliang Li. 2018. Approximate query processing: What is new and where to go?. In *Data Science and Engineering*. Springer, 379–397.
[30] S.L. Lohr. 2010. *Sampling: Design and Analysis*. Brooks/Cole.
[31] Barzan Mozafari and Carlo Zaniolo. 2010. Optimal load shedding with aggregates and mining queries. In *ICDE*. IEEE, 76–88.
[32] Christopher Mutschler et al. 2013. The DEBS 2013 Grand Challenge. In *DEBS*. ACM, 289–294.
[33] C. Mutschler and M. Philippsen. 2013. Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams. In *IPDPS*. IEEE, 1133–1144.
[34] Do Le Quoc et al. 2017. StreamApprox: approximate computing for stream analytics. In *Middleware*. USENIX, 185–197.
[35] Nicoló Rivetti et al. 2016. Load-Aware Shedding in Stream Processing Systems. In *DEBS*. ACM, 61–68.
[36] Nicolo Rivetti et al. 2018. Probabilistic management of late arrival of events. In *DEBS*. ACM, 52–63.
[37] Anshumali Shrivastava et al. 2016. Time adaptive sketches (ada-sketches) for summarizing data streams. In *SIGMOD*. ACM, 1417–1432.
[38] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *PODS*. ACM, 263–274.
[39] Michael Stonebraker et al. 2005. The 8 Requirements of Real-Time Stream Processing. In *SIGMOD*. ACM, 42–47.
[40] Nesime Tatbul et al. 2003. Load shedding in a data stream manager. In *PVLDB*. ACM, 309–320.
[41] Nesime Tatbul et al. 2007. Staying fit: Efficient load shedding techniques for distributed stream processing. In *PVLDB*. ACM, 159–170.
[42] Ankit Toshniwal et al. 2014. Storm at twitter. In *SIGMOD*. ACM, 147–156.
[43] N. Zacheilas et al. 2015. Elastic complex event processing exploiting prediction. In *BigData*. IEEE, 213–222.
[44] Nikos Zacheilas et al. 2017. Maximizing determinism in stream processing under latency constraints. In *DEBS*. ACM, 112–123.
[45] Matei Zaharia et al. 2016. Apache Spark: A Unified Engine for Big Data Processing. In *Communications of the ACM*, Vol. 59. ACM, 56–65.